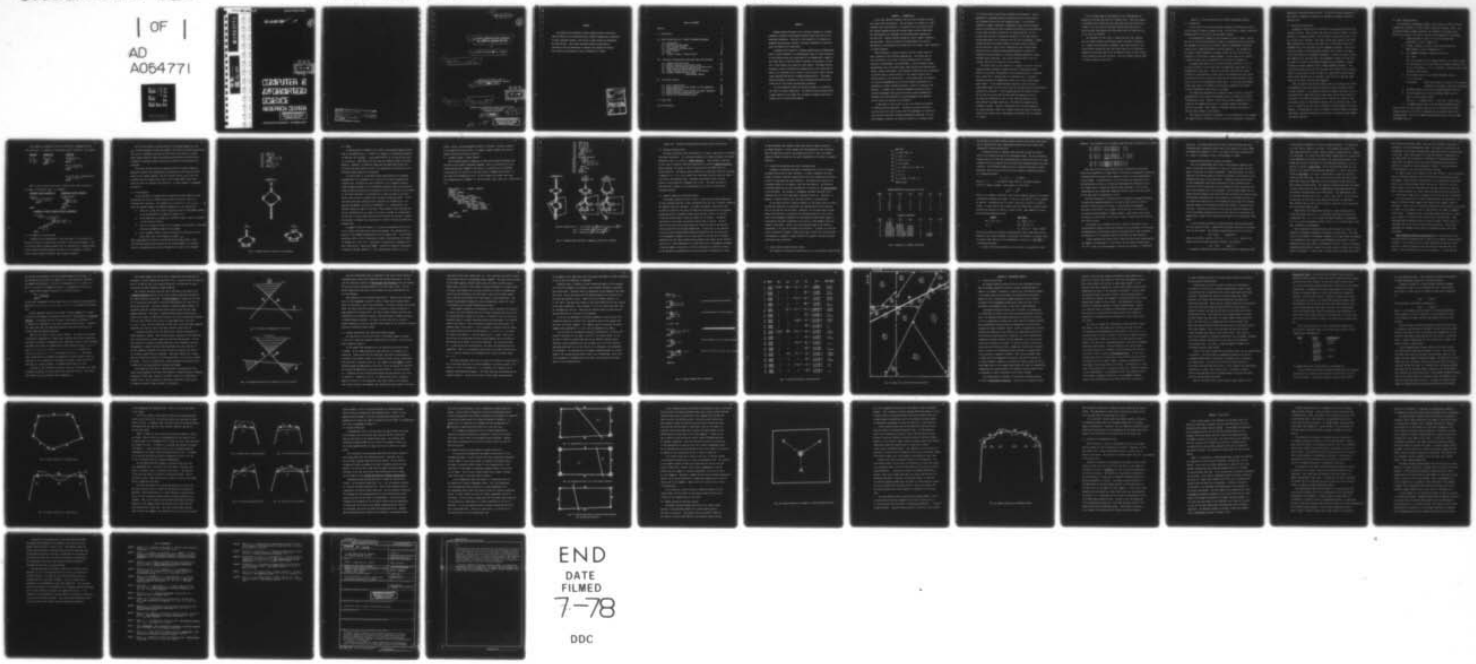
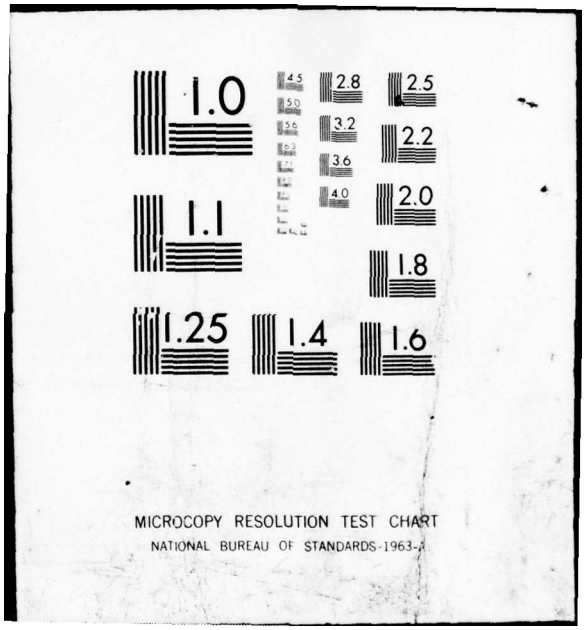
AD-A054 771 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 9/2
A FINITE DOMAIN-TESTING STRATEGY FOR COMPUTER PROGRAM TESTING.(U)
AUG 77 E I COHEN, L J WHITE AFOSR-77-3416

UNCLASSIFIED OSU-CISRC-TR-77-13 AFOSR-TR-78-1014 NL

| OF |
AD
A054771



END
DATE
FILMED
7-78
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(2)

COMPUTER & INFORMATION SCIENCE RESEARCH CENTER

THE OHIO STATE UNIVERSITY COLUMBUS, OHIO

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DDC
This technical report has been reviewed and is approved for public release IAW AFR 190-12 (7b).
Distribution is unlimited.
A. D. BLOSE
Technical Information Officer

(14) OSU-CISRC-TR-77-13

(2)

(6) A FINITE DOMAIN-TESTING STRATEGY FOR COMPUTER PROGRAM TESTING.

by

(10) Edward I. Cohen and Lee J. White

(9) Interim rept.,

Research Supported in Part by
Air Force Office of Scientific Research
(15) AFOSR [redacted]-77-3416

(18) AFOSR (19) TR-78-1414

Computer and Information Science Research Center
The Ohio State University
Columbus, Ohio 43210

(12) 58 p.

(11) Aug[redacted]977

407 586

PREFACE

The Computer and Information Science Research Center of The Ohio State University is an interdisciplinary research organization consisting of staff, graduate students, and faculty of many University departments and laboratories. This report describes research undertaken in cooperation with the Department of Computer and Information Science.

This research was supported in part by AFOSR Grant 77-3416.

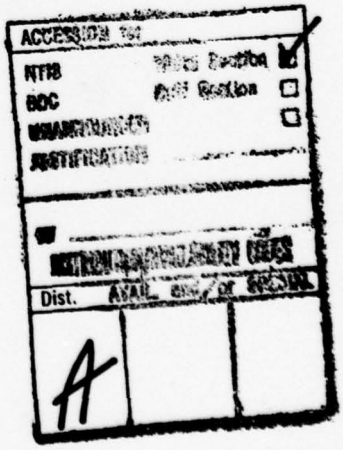


TABLE OF CONTENTS

ABSTRACT

Program testing continues to be a practical approach to software validation, however the strategies currently being used lack a solid analytical foundation. The goal of this research is to analyze the program testing process, develop a strategy to maximize its effectiveness, and identify its limitations.

A program can be viewed as a complex mapping from an N-dimensional space of input variables to an M-dimensional space of output variables. In the testing process the correctness of the program over a domain of this input space is inferred from its observed correctness on a small set of "well-chosen" test values for that domain. The Domain Testing Strategy is used to determine the necessary set of test values and is shown to be successful for all types of errors except a small subclass called "Missing Path Errors of Reduced Dimensionality". The domain testing strategy is developed for both continuous and discrete input spaces and for both linear and nonlinear predicates.

The only completely effective testing strategy is an exhaustive test which is totally impractical. The domain testing strategy offers a major reduction in the high cost of computer program testing with a minimal loss of testing effectiveness.

CHAPTER I. INTRODUCTION

At the same time that hardware costs have been dropping, software costs have risen precipitously. The development of software is a labor intensive task, and therefore it is inherently an error-prone process. The software systems we need have become larger and more complicated, but unfortunately our techniques for developing these systems have not kept pace. Typical software systems are overly complex, expensive, and unreliable. Unless we develop new methodologies for creating, maintaining, and analyzing these systems they will become a major impediment to further progress.

There continues to be much research activity in many aspects of what we generally term "Software Engineering". The overall goal of this discipline is to be able to develop software which is correct, efficient, understandable, and maintainable. Various software design methodologies such as top-down design, HIPO (hierarchical-input-process-output), Stepwise Refinement, and Information Hiding are being investigated as possible ways to bring discipline to the usually chaotic task of software design at the system level. At the coding level the concepts of structured programming and structured program documentation are attempting to produce program code which is easier to understand and maintain. In another area, software management techniques such as the Chief Programmer Team concept and various system documentation standards are being developed to provide management with improved control during the lifetime of the software.

A large part of the activity in this area concerns the problem of "Software Quality Assurance". Software is and will continue to be error-prone, but there is much room for improvement in reliability. There has been some work in"Software Reliability Modeling", but this work attempts to predict the expected reliability of software based

on its error history rather than to improve the reliability. The two approaches to improving software reliability which are being studied are "Program Verification" and "Program Testing". In verifying a program we attempt to generate a mathematical proof that the program correctly transforms the input variables described by an input assertion to the output variables described by an output assertion. At the current time this goal is beyond the capability of our best theorem proving algorithms, and the prospects are not very encouraging. Program testing has been the standard method of software quality assurance, but the scientific study of this technique has just recently begun. We believe that a program testing methodology which is based on a rigorous scientific analysis is the most promising approach to achieve software quality assurance. This report describes some of the preliminary results of our research in this problem.

The overall goal of the research is to replace the ad hoc intuitive program testing strategies currently used with a methodology firmly based on a scientific investigation of the testing process, to determine why it works and to identify what its limitations are. In program testing we execute the program with a small set of well-chosen sample input values. If the program produces incorrect results for any of these sample inputs an error exists, and the testing process has successfully indicated its existence. If the program produces the desired results for all the test inputs then we infer that the program is correct. Of course the confidence we place in this inference depends on how "well-chosen" the sample inputs are. We define the program testing problem as follows: "Given no information other than the program to be tested, generate a small set of sample input data which if processed correctly will ensure with a high degree of confidence that the program is correct".

In the current phase of the research we are investigating the problem of selecting test data for a specific path. Using the sequence of statements which constitutes the path being tested we characterize a small set of test data for the path, evaluate how effectively it tests the path, and identify any errors which won't be found with the set of test data selected.

The remainder of this report is broken down into four chapters. The next chapter defines and discusses some of the basic concepts we use. Chapter three describes a program's input space structure, and in particular characterizes the input space region for a single path. Chapter Four discusses some of the preliminary results we have obtained in the problem of test data selection. The last chapter outlines plans for future research in this area.

Chapter II - Basic Definitions and a Sample Programming Language

2.1 Introduction

In this chapter we introduce some basic concepts and definitions central to the problem of computer program testing. We also define a simple programming language which will be used in writing sample programs.

In designing this language we have attempted to keep it simple, concise, and easy to use while being powerful enough to program non-trivial algorithms. The language is structured and contains a general IF-THEN-ELSE alternation construct and a powerful DO WHILE iteration capability. The language does not contain a GOTO statement or statement labels, but this has not been a hindrance in the writing of sample programs. At this stage of the research a single real data type is sufficient for our purposes of exposition.

A program which reads a set of N input variables and writes a set of M output variables is said to map from an N-dimensional input space to an M-dimensional output space. Each set of specific values for the N input variables is represented as a single point in the N-dimensional input space, and similarly each set of specific values for the M output variables represents a point in the M-dimensional output space. In this model the N-dimensional input space is the domain of the function computed by the program, and the M-dimensional output space is its range.

The computation performed by a program is usually viewed as a function mapping points in the domain (N-dimensional space) to points in the range (M-dimensional space). However, in this work we view a program as a set of functions, each mapping the points in a specific region of the domain to points in the range. These regions are mutually exclusive and exhaustive over the entire domain of the program.

Each region and function corresponds to an executable path in the program. The region itself is defined by the set of predicate constraints (conditional

expressions) encountered along the path. The specific function computed for each region is formed by the sequence of assignment statements encountered along the path.

2.2 Variables and Data Types

A program's variables will be divided into three mutually exclusive and exhaustive classes. Each variable will either be an input variable, an output variable, or a program variable. We have placed certain restrictions on the use of each type of variable. Each input variable must appear in the single READ statement and in addition can appear only in predicates (both IF and DO WHILE) and on the right hand side of assignment statements. Output variables must appear in the single WRITE statement and in addition can appear on the left hand side of assignment statements. In fact each output variable must appear on the left hand side of an assignment statement on every execution path or its value at the WRITE statement will be undefined. Program variables can appear anywhere except in READ and WRITE statements. This classification and these rules will help make later analyses and discussions easier to follow since an input variable will never have its value changed, and an output variable will be assigned a value only once, when the computation of the output value has been completed.

There is only one type of data element, and it will contain real values. The language includes both simple variables and single subscripted arrays of these variables. In an idealized theoretical machine the choice of real-valued data would imply that both the input space and the output space are continuous. This would be desirable since we could be sure that any test data points we want to characterize actually exist in the input space. However, in reality we are only able to represent a discrete subset of the real numbers. In discussing our results in Chapter 4 first we assume a continuous input space, and then we analyze how the reality of a discrete space changes the results obtained.

2.3 Basic Program Elements

The experimental programming language I have chosen is a well-structured language which consists of a small number of powerful statement types. The following statement elements constitute the statement and statement parts which are the basis of the control structure of any program in the language.

1) A READ statement in either of the two following forms:

   READ (VIN(I), I=1,N) or READ I,J,K,...

2) A WRITE statement in either of the two following forms:

   WRITE (VOUT(I), I=1,M) or WRITE M,N,O,P,...

3) An arithmetic assignment statement of the form:

   Var. = Expr.

   where the variable can be a program variable or an output variable and the expression is the usual type of arithmetic expression over the set of operators (+,-,*,/), parentheses, variables except output variables, and constants.

4) The IF (Pred.) part of the IF THEN ELSE ENDIF construct.

   It is of the form:

   IF predicate where the predicate in general can be a boolean combination of arithmetic relational expressions.

5) The ENDIF delimiter part of the IF THEN ELSE ENDIF construct.

6) The DO statement part of the DO (DO-body) ENDDO construct.

   There are three possible forms:

   DO (iter.), DO WHILE (pred.), or DO (iter.) WHILE (pred.).

7) The ENDDO delimiter part of the DO (DO-body) ENDDO construct.

There are also two other elements which will not be considered as program elements because we don't have to represent them as separate nodes in the program digraph. They are the THEN and ELSE delimiter parts of the IF THEN ELSE ENDIF construct.

So in general a program will have the three basic programming control flow constructs: 1) sequence, 2) alternation, and 3) iteration. For example:

SEQUENCE

```
A = 1;
B = A+2;
C = A-B;
```

ALTERNATION

```
IF A = B
  THEN C = 1;
  ELSE C = 2;
ENDIF;
```

ITERATION

```
DO WHILE (A> 0);
  A = A-1;
ENDDO;
```

```
DO I = 1 TO 10 BY 1;
   X(I) = X(I) + 1;
ENDDO;
```

```
DO I=1 TO 10 BY 1 WHILE(X(I)>0);
   X(I) = X(I) + 1;
ENDDO;
```

Each of these structure types may be nested within other structures of the same or of different types. For example:

ITERATION WITHIN ALTERNATION

```
IF A = B
  THEN DO I = 1 TO 10 BY 1;
          X(I) = X(I)+1;
       ENDDO;

ENDIF;
```

ALTERNATION WITHIN ITERATION

```
DO WHILE (A>0);
  IF B = 0
     THEN B = A-1;
     ELSE B = A+1;
  ENDIF;
  A = A-C;
ENDDO;
```

ALTERNATION WITHIN ITERATION WITHIN ALTERNATION

```
IF A = B
     THEN DO I = 1 TO 10 BY 1;
               IF X(I)>0
                    THEN X(I) = X(I) + 1;
                    ELSE X(I) = X(I) - 1;
               ENDIF;
          ENDDO;
     ELSE A = C;
          B = C;
ENDIF;
```

2.4 A Program as a Directed Graph

A program will be represented as a directed graph G = (V,A) where V is the set of vertices or nodes and A is the set of arcs (directed edges). Each of the seven types of program elements will be represented by a separate node in the original unreduced digraph, and each arc will represent a possible flow of control between individual nodes (program elements).

We will also define two reduced forms of the program digraph, G_{r1} and G_{r2}, in which sequences of nodes are merged if they must be executed sequentially as a block of statements each time the first node of the block is reached. These reduced forms will make the processing and analysis more efficient by reducing the information about nodes and arcs which must be maintained and processed.

The second reduced form G_{r2} is generated from the original digraph by making all possible node mergings while maintaining all the flow paths which exist in the original digraph. The first reduced form G_{r1} uses a more conservative rule for merging nodes since it won't merge nodes in some cases which are parts of different flow constructs. A simple example is diagrammed in Figure 1.

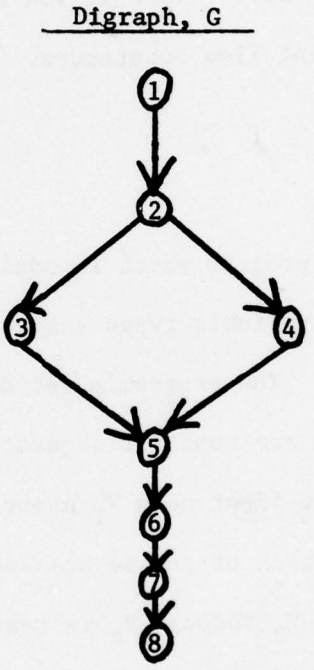
2.5 Valid Programs

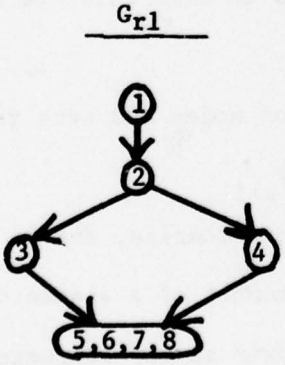
A valid program is a program which in addition to being compiled cleanly and meeting the rules for variable types stated in section 2.2 meets the following four conditions. The program's set of nodes is $V = (V_1, V_2, V_3, \ldots, V_k)$ where the program elements are numbered sequentially in the program text.

1) There is a unique first node V_1 associated with the only READ statement in the program which of course has indegree zero.
2) Every other node V_2 through V_k is reachable along at least one sequence of nodes and arcs from V_1.
3) There is a unique last node V_k with outdegree of zero which is associated with the only WRITE statement in the program.
4) V_k is reachable along at least one sequence of nodes and arcs from every other node in the program V_1 through V_{k-1}.

These requirements will make our further definitions more concise, and it will make any analysis easier because we can assume somewhat of a standard form for a program without any loss of generality or power in our language.

```
(1)   READ A,B;
(2)   IF A = B
(3)     THEN C = A + B;
(4)     ELSE C = A - B;
(5)   ENDIF;
(6)   D = 2*C - 2;
(7)   E = C + A - B;
(8)   WRITE D,E;
```





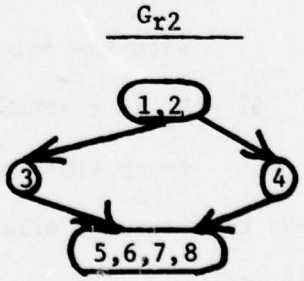


Fig. 1: Program (without iteration) and digraphs.

2.6 Paths

A "control path" is defined to be a walk in the program digraph starting with V_1 and ending with V_k. A "walk" in a digraph is an alternating sequence of nodes and arcs $V_1A_1V_2A_2 \ldots A_{k1}V_k$ where each arc A_i is directed from node V_i to node V_{i+1}. Each node or arc can occur any number of times in the walk sequence. Therefore, it should be noted that two walks which differ only in the way that they execute a loop in the program will be defined to be two different control paths for our purposes.

A "control path" is a path which exists syntactically in the digraph representation of the program, but it may not be an actual executable and testable path. We define the "path condition" to be a compound predicate formed by ANDing together all the individual predicates which are encountered along the path and which must be satisfied in order for the path to be followed. If this "path condition" is satisfiable by at least one value of the input vector, then the path is feasible and is called an "execution path". If the individual predicates forming the path condition are contradictory or not mutually satisfiable, the path is "infeasible", i.e., nonexecutable, and therefore cannot be used as a test path. This problem of path infeasibility will be troublesome since at times we will have to change our strategy when we find that a "control path" which has been selected for testing is infeasible and must be replaced by an equivalent (in terms of testing desirability) feasible path.

A program's "path set formula" is a concise representation for the set of all control paths which exist within the program. The representation is similar to the standard representation used for regular sets. In the formula sequential control flow is represented by simple ordering within the formula, for example st-1, st-2, st-3. Alternation is represented by something like Pred. (then-action + else-action) ENDIF. Iteration is generally represented as DO-Pred. (DO-body, DO-Pred.)*. This representation allows us to use a

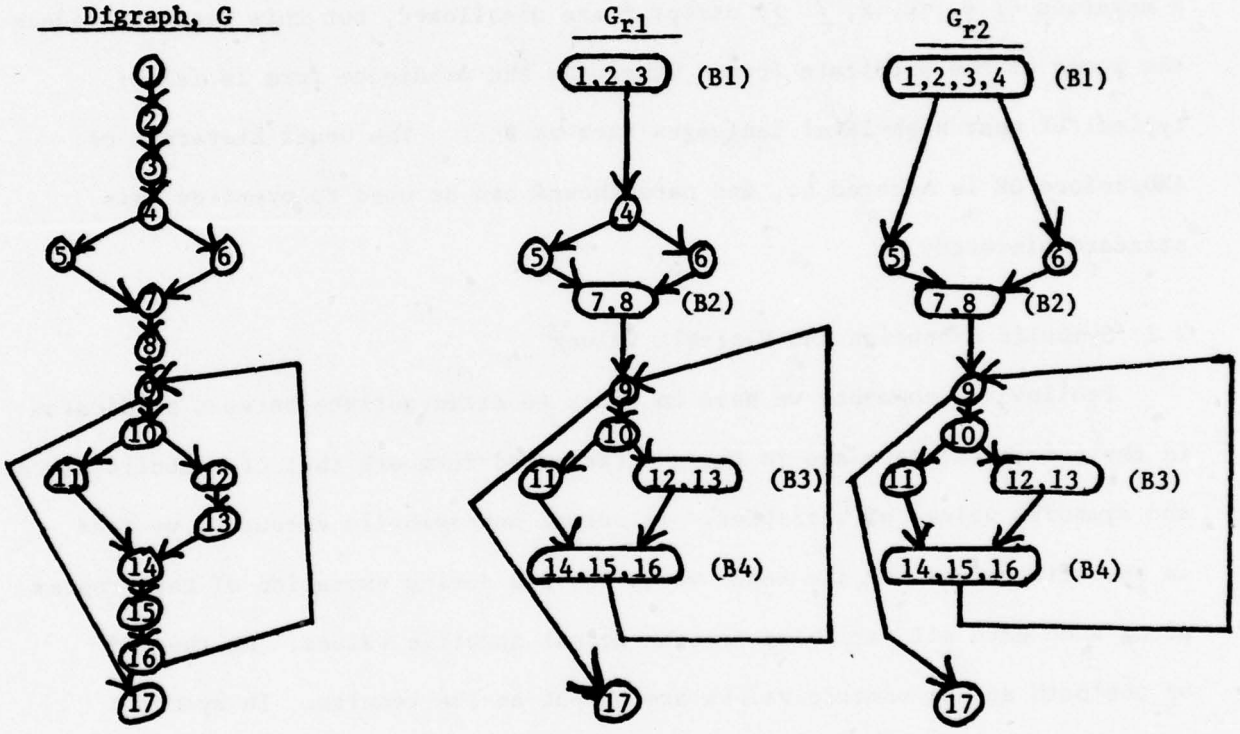
finite, concise, and understandable formula to represent a possibly infinite set of complicated control paths. An example program, digraph, and path set formula follow are provided in Figure 2.

2.7 A Sample Program - "Binary Search"

The following program to implement a binary search algorithm demonstrates the power of the sample programming language. XIN is a sorted vector of the values searched over; VALSRCH is the value to be searched for; and VALFIND is the subscript or position of the value equal to VALSRCH found in XIN. If the search fails VALFIND will be Ø. In this example the integer input space has a dimensionality of SUBMAX + 1, and the integer output space has a dimensionality of 1.

```
READ (XIN(I), I = 1, SUBMAX), VALSRCH;
LOBOUND = 1;
HIBOUND = SUBMAX;
VALFIND = 0;
DO WHILE (HIBOUND ≥ LOBOUND);
     MIDVAL = (HIBOUND + LOBOUND)/2;
     IF XIN(MIDVAL) > VALSRCH
          THEN HIBOUND = MIDVAL - 1;
          ELSE IF XIN(MIDVAL) < VALSRCH
                    THEN LOBOUND = MIDVAL + 1;
                    ELSE VALFIND = MIDVAL;
                         HIBOUND = LOBOUND - 1;
               ENDIF;
     ENDIF;
ENDDO;
WRITE VALFIND;
```

```
(1)   READ A,B,C;
(2)   V1 = A + B;
(3)   V2 = B + C;
(4)   IF V1 < V2
(5)     THEN V3 = 0;
(6)     ELSE V3 = 1;
(7)   ENDIF;
(8)   V4 = V3;
(9)   DO WHILE (V4 > 0);
(10)    IF A < B
(11)      THEN V4 = 0;
(12)      ELSE V5 = V2 - V1;
(13)           V4 = V4 + 1;
(14)    ENDIF;
(15)    V6 = 2*V5;
(16)  ENDDO;
(17)  WRITE V6;
```



Path Set Formula for G = $1,2,3,4(5+6)7,8,9\ [10(11+12,13)14,15,16,9]^*17$

G_{r1} = $B1,4(5+6)B2,9\ [10(11+B3)B4,9]^*17$

G_{r2} = $B1(5+6)B2,9\ [10(11+B3)B4,9]^*17$

Fig. 2: Program (with iteration), digraphs, and path set formulae

Chapter III. Predicate Interpretations and Input Space Partitioning

3.1 Standard Predicate Forms

The general predicate form utilized here is a logical combination of arithmetic relational expressions. If a predicate consists of a single arithmetic relational expression, then it is called a Simple Predicate. Any predicate consisting of two or more arithmetic relational expressions is called a Compound Predicate. These predicates can appear in both IF constructs and the WHILE option of the DO construct. The logical operator NOT and all relational operators including a negation (i.e., $\not<$, $\not>$, $\not\leq$, $\not\geq$) except $\neq$ are disallowed, but this does not reduce the power of the predicate form. Otherwise the predicate form is fairly typical of most high-level languages such as PL/I. The usual hierarchy of AND before OR is adhered to, and parentheses can be used to override this standard hierarchy.

3.2 Symbolic Execution and Variable Values

Preliminary concepts we need in order to differentiate between predicates in their original form and in their interpreted form are that of symbolic execution and symbolic values of variables. In normal non-symbolic execution we read in specific values for the input variables and during execution of the program along some path all variables contain actual specific values. At the end of the path actual numeric values are output as the results. In symbolic execution we first choose the specific path we want to execute. We don't have to find actual input values which will cause this path to be executed since the path will be executed symbolically. At any point in the execution of the path each program variable which has appeared on the left hand side of an assignment statement has its value stored in the form of an arithmetic expression over the input variables and constants since these are the original source of all values appearing on the right hand sides of assignment statements. In essence symbolic execution of a path is an execution in which the values of all variables are maintained in the form of symbolic expressions in terms

of input variables and constants rather than specific numeric values as in normal execution. A short example will help demonstrate this difference. Since we symbolically execute any program one path at a time, the example program in Figure 3 needs only one path to demonstrate the concept of symbolic execution.

3.3 Original Form Predicates and Their Interpretations

In general a predicate constraint is expressed over the set of program variables and the set of input variables. In order to generate test data values to execute a specific program path we have to transform each predicate so that it constrains only input variables. We replace each occurrence of a program variable by its symbolic value and then simplify. We define the constraint imposed on the input variables by a predicate as the Interpretation of the predicate. The sequence of assignment statements executed can be different for each of the many paths a predicate occurs on. Therefore, in general, a single predicate will have many different interpretations.

If two paths on which a predicate appears are the same up until the point at which the predicate is encountered, the interpretation of the predicate will be the same for the two paths. Similarly, for two paths which diverge before the predicate is encountered the interpretations will still be the same if the assignment statements for the program variables which appear in the predicate and their predecessors are totally in code segments which are common to both paths. Lastly, the interpretations might be the same by a coincidence in the way the variables are calculated. It should be noted that since a predicate can appear on many paths, some of which will diverge before the predicate is reached, a single predicate can appear to be many different constraints when interpreted.

3.4 Linear and Non-linear Predicate Forms

The linearity of predicate interpretations is a very important characteristic.

```
          READ A,B;
(1)
          C = 2*A + 3*B - 4;
(2)
          D = 2*C - 2;
(3)
          E = D/2 + A + 1;
(4)
          C = (E + D)*3;
(5)
          F = C - 6*D + E + A + 6*B - 14;
(6)
          D = D - C;
(7)
          G = E + 3;
(8)
          H = D + 6*E - A + 3*B - 7;
(9)
          WRITE F,G,H;
```

REGULAR EXECUTION (with inputs A=2, B=1)

pt.	C	D	E	F	G	H
1	-	-	-	-	-	-
2	3	-	-	-	-	-
3	3	4	-	-	-	-
4	3	4	5	-	-	-
5	27	4	5	-	-	-
6	27	4	5	2	-	-
7	27	-23	5	2	-	-
8	27	-23	5	2	8	-
9	27	-23	5	2	8	1

SYMBOLIC EXECUTION

pt.	C	D	E	F	G	H
1	-	-	-	-	-	-
2	2A+3B-4	-	-	-	-	-
3	2A+3B-4	4A+6B-10	-	-	-	-
4	2A+3B-4	4A+6B-10	3A+3B-4	-	-	-
5	21A+27B-42	4A+6B-10	3A+3B-4	-	-	-
6	21A+27B-42	4A+6B-10	3A+3B-4	A	-	-
7	21A+27B-42	-17A-21B+32	3A+3B-4	A	-	-
8	21A+27B-42	-17A-21B+32	3A+3B-4	A	3A+3B-1	-
9	21A+27B-42	-17A-21B+32	3A+3B-4	A	3A+3B-1	1

Fig. 3: Regular vs. Symbolic Execution

The testing of non-linear borders requires many more test points than linear, and the computational task of generating solutions for sets of non-linear constraints is much more difficult.

A "Predicate Form" is a general term referring to both the original form of the predicate and any of the interpreted forms. The only difference between these two is that the interpreted form is restricted to only the input variables while the original form can contain both program variables and input variables. The following standard form for a predicate describes a simple predicate, which is also the form for each arithmetic relational expression of a compound predicate.

$$T_1 + T_2 + \ldots\ldots + T_L \text{ ROP } K$$

where T_i $i = 1, \ldots, L$ are general terms, ROP is a relational operator, and K is a numeric constant. Each Term T_i is of the form

$$A_i X_1^{P_i} \ldots X_N^{P_N}$$

where A_i is a numeric coefficient, the X's are variables and the P's are powers to which the variables are raised. If any P_i is zero the $X_i^{P_i}$ will in general be omitted. A predicate form is linear if and only if each of its terms is linear. A linear term is one in which only one variable has a non-zero power, and this one variable is raised to a power of one. Some examples of linear and non-linear predicates follow.

LINEAR	NON-LINEAR
$3X_1 - 2X_2 + X_4 \leq 7$	$X_1 - 2X_2X_3 + X_4 > 3$
$X_2 = 3$	$3X_1 + X_3^3 - X_4 < 2$
	$X_2 + X_3/X_4 \geq 1 \quad (X_3/X_4 = X_3X_4^{-1})$

As we can see above any predicate containing one or more terms which are the product of two or more variables, the quotient of variables, a variable raised to a power other than one, or any combination of these (e.g., $X_1^2X_2/X_3^3X_4^2$) is a non-linear predicate.

At times a non-linear predicate can be reduced to an equivalent linear

predicate. The following list of examples shows what type of reductions are allowed.

(1) IJ/J = 2 can be reduced to I = 2.
(2) IJ/J ≠ 2 can be reduced to I ≠ 2.
(3) IJ/J ≤ 2 can be reduced to I ≤ 2.

(4) I/J = 3 can be reduced to I - 3J = 0.
(5) I/J ≠ 3 can be reduced to I - 3J ≠ 0.
(6) I/J ≤ 3 cannot be reduced to I - 3J ≤ 0.

(7) IJ - J = 0 cannot be reduced to I = 1.
(8) IJ - J ≠ 0 cannot be reduced to I ≠ 1.
(9) IJ - J ≤ 0 cannot be reduced to I ≤ 1.

From this list we see that whenever we try to multiply or divide both sides of an inequality by a common factor in the form of an expression containing a variable which can be positive or negative (cases 6 & 9) the transformation is not allowed. In addition cases 7 & 8 show us that the properties of the multiplicative zero force us to disallow transformations involving division by a common factor in equalities and not-equals predicates. All the other transformations are allowed even though the solution sets of the forms differ at the point where the demoninator is zero.

The above discussion of linearity must be extended to compound predicates. In general any compound predicate of the form C_1 AND C_2 will be linear if and only if both C_1 and C_2 are linear. Also in general any compound predicate of the form C_1 or C_2 changes the solution set to two sets which can contain variable numbers of common elements. Again both C_1 and C_2 must be linear, but in this case if only one is linear at least one of the solution sets will be linear.

A predicate in its original form can be linear or non-linear, but in testing specific paths we really are interested in the linearity of the various interpretations of the predicate since the interpretation is what determines how difficult it is to generate effective test data for that predicate.

Linear predicates can easily lead to linear and non-linear interpretations. For example, the predicate C < 2 can lead to the non-linear interpretation $A^2 - B^2 < 2$ when C has been assigned a value by the statement C = (A+B)*(A-B).

Similarly, non-linear predicates can lead to both non-linear and linear interpretations, although the latter case is not expected to be very common. For example the predicate $D > B^2 + 3$ can lead to the linear interpretation $A > 3$ when D is assigned a value by the statement D = B*B+A.

3.5 General Characteristics of Borders and Regions

The set of program paths which exist in the program's digraph, represented by the path set expression, is the set of control paths, but only some of these paths are actually executable. Those paths which are control paths but not execution paths will never be followed in the execution of the program by any value of the input vector. Therefore, there are no points in the N-dimensional input space associated with these infeasible paths. There will be one or more points in the N-dimensional space associated with each of the execution paths of the program. Since we have assumed that all abend errors and infinite loop errors have been discovered and corrected using other testing techniques prior to the running of our testing procedure, we are assured that every point in the N-dimensional input space will complete execution along one of the execution paths and therefore is associated with that path. Obviously each point will only execute along one program path. So the relationship between paths and the input space is a one-to-many onto mapping from a subset of the program control paths to the N-dimensional input space.

In the following discussion we assume predicate interpretations which are both simple and linear. Compound and non-linear forms will be discussed later in this chapter. Any predicate interpretation will be in the form

$$A_1X_1 + A_2X_2 + \ldots + A_NX_N \text{ ROP } B,$$

where the relational operator will be $\leq$, $<$, $\geq$, $>$, $=$, or $\neq$. However, the border which any of these predicates defines is the equality

$$A_1X_1 + A_2X_2 + \ldots + A_NX_N = B.$$

In a general N-dimensional space this equality defines a hyperplane, which

is the geometric equivalent of a plane in spaces where N=3. In 2-space this is a line, and in 1-space it is a point. So in general the border in the input space produced by a simple, linear predicate interpretation is a segment of a hyperplane. Of course, the border itself, namely the set of points lying on the hyperplane may or may not be part of the region being defined. The border will be part of the region for the relational operators $\leq$, $\geq$, and =, and this is called a closed border. On the other hand the border will not be part of the region for the relational operators <, >, and $\neq$, and this is called an open border. Of course, a border which is open for one path is closed for some other path. This can be seen in the fact that when we have paths which execute the THEN option of an IF construct we also have paths which execute the ELSE option. The predicate interpretation which will appear on these latter paths will have the complementary relational operator from the predicate interpretation on the former paths. The complementary pairs of relational operators are $\leq$ and >, $\geq$ and <, and = and $\neq$. We can see that in each pair one will form an open border and the other a closed border. This obviously must be true since we know that each point in the input space must follow some execution path.

Geometrically, the region which satisfies a single predicate with a relational operator of $\leq$ or $\geq$ is a closed half-space which is bordered by a hyperplane; for < or > it is an open half-space bordered by a hyperplane; for = it is only the hyperplane itself; and for $\neq$ it is two half-spaces, both open, bordered by the hyperplane.

Each execution path considered apart from the program as a whole is a sequence of assignment statements and predicate interpretations. The composition of the sequence of assignment statements on the path defines the mapping or transformation from a region of the N-dimensional input space to the M-dimensional output space which is performed on the path. The set of predicate interpretations defines the region of the input space over which this transformation is performed. In other words it defines the set of input points which

will follow this particular path. This set can be as small as one point or as large as the entire N-dimensional space. The region for each path is the intersection of the regions satisfying the individual predicate interpretations on the path. Since these regions are either half-spaces or hyperplanes, we are talking about the intersection of half-spaces or hyperplanes,which are convex sets, the regions of the input space associated with the execution paths will in general be convex polytopes in N-space. This characteristic will be important in our test data generation strategy.

Of course, there is one exception to the above analysis. Since a predicate with the relational operator $\neq$ will be satisfied by points forming two open half-spaces, a path with any $\neq$ predicates on it can potentially be associated with a set of convex polytopes rather than just one. This will not pose a major problem for the test data generation strategy, since we can view this set of convex polytopes as a single convex polytope with slices missing from it. This allows a common basic strategy to be used for any set of predicates.

One last characteristic of these regions which will be important is the number of different border segments which makes up the complete border of the region. There will be one constraint generated for each predicate encountered on a path, but this number is only an upper bound on the number of segments in the border. A predicate within a loop might appear many times on a path, but of course all occurences other than the first are superfluous. Some predicates when interpreted reduce to relations between constants which are tautologies and do not further constrain the solution set. Also certain predicates are redundant because they are superceded by more restrictive constraints.

Input Independent Predicate Interpretations are those predicate interpretations which are expressed only in terms of an arithmetic relation between constants. This relation is true or false regardless of what input values are used to reach the predicate along the particular path which has been traced. If the interpretation is true, for example $5 > 3$, then this does

not add any new constraint to the set already generated for the path. If the interpretation is false, for example $5 \leq 4$, then the path is infeasible. For example the predicate C > A+2 will be interpreted as 0 > -2 when C = 5 and A = 1 or when C is assigned the value A + 4 regardless of the value of A. Another more frequent occurrence is in a DO loop such as

```
DO I = 1, 5, 1;
     --LOOP BODY--
ENDDO;
```

In this case, regardless of input values, the set of predicate interpretations generated is $1 \leq 5$, $2 \leq 5$, $3 \leq 5$, $4 \leq 5$, $5 \leq 5$, $6 > 5$, and these are obviously true.

Another important reason why the number of border segments for a region is less than the number of predicates on the path is the phenomenon of Predicate Dominance. A constraint will not form part of the region's boundary if it is implied by a combination of the other constraints for the path. In general this comes about when the dominating predicates form more restrictive constraints than the dominated predicate.

There are two types of cases we must consider: the simple case in which a predicate is dominated by a single other predicate and the more complicated case where a predicate is dominated by the combination of two or more predicates. The former case exists for example when we have constraints such as $J \leq I + 2$ and $J \leq I + 4$ or $J \leq I + 2$ and $J = I + 1$. If $J \leq I + 2$, it obviously is also $\leq I + 4$; also if $J = I + 1$ it obviously is also $\leq I + 2$. The latter case is both harder to recognize and expected to be more frequent since the solution set of a combination of predicates in general is more restrictive than that of a single predicate. An example should make this clear.

In Figure 4, the predicate C_3 does not form part of the border for region R because (C_1 AND C_2) is a more restrictive constraint than C_3. Also notice that neither C_1 nor C_2 taken by itself dominates C_3.

This simple example can also be used to demonstrate path infeasibility. In Figure 5, we have reversed the direction of C_3. We can see that the solution sets of (C_1 AND C_2) and C_3 are mutually exclusive, and therefore no points can satisfy the path condition (C_1 AND C_2 AND C_3).

The concepts described above must now be extended to the general case. A Compound Predicate consists of two or more simple predicates connected by the logical operators OR and AND. The Path Condition is defined as the total set of constraints which must be satisfied simultaneously for input data to follow the given path. The path condition is formed by ANDing together the individual predicate interpretations found along the path into one large compound predicate. Therefore, a compound predicate whose interpretation along a path contains an AND logical operator will cause no additional problems since the path condition will still be a single ANDed term of the form $C_1 \wedge C_2 \wedge \ldots \wedge C_N$. The only difference in this case is that the single compound constraint will add two terms to the path condition rather than one.

A compound predicate with an OR defines two, possibly unconnected and widely separated, regions. Both of these regions correspond to the particular path, and any points in either region will execute along the path. For the path condition in disjunctive normal form each of the elementary conjuncts corresponds to a region, and any points satisfying one, more than one, or all the elementary conjuncts will follow the path in question, but some or all of these conjuncts may be infeasible. These many regions for a single path can be mutually exclusive, can overlap partially, or one can be a subset of another. We have to consider each of these possibilities and decide how to adjust our test data generation strategy accordingly.

Even though the only type of compound predicate interpretation which causes extra problems is the OR'ed type, whenever a program contains a compound predicate of either type, there will be paths on which the interpretation contains on OR. This is because on some paths a predicate C_1 AND C_2 must be negated, and NOT (C_1 AND C_2) $\equiv$ (NOT C_1) OR (NOT C_2).

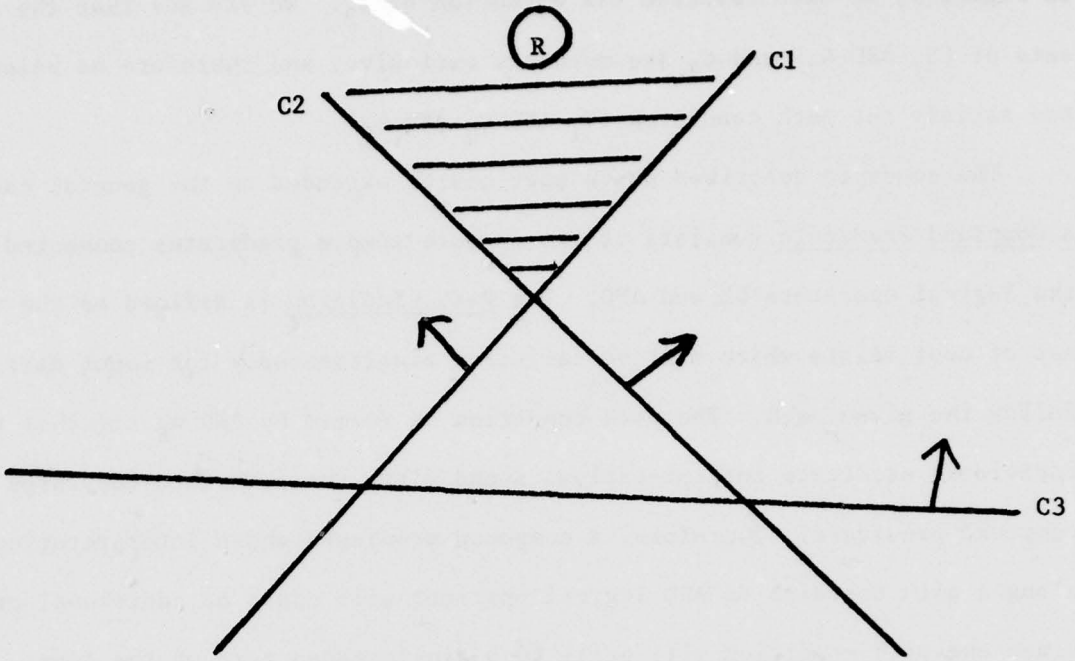


Fig. 4: Predicate C3 Dominated by (C1 and C2)

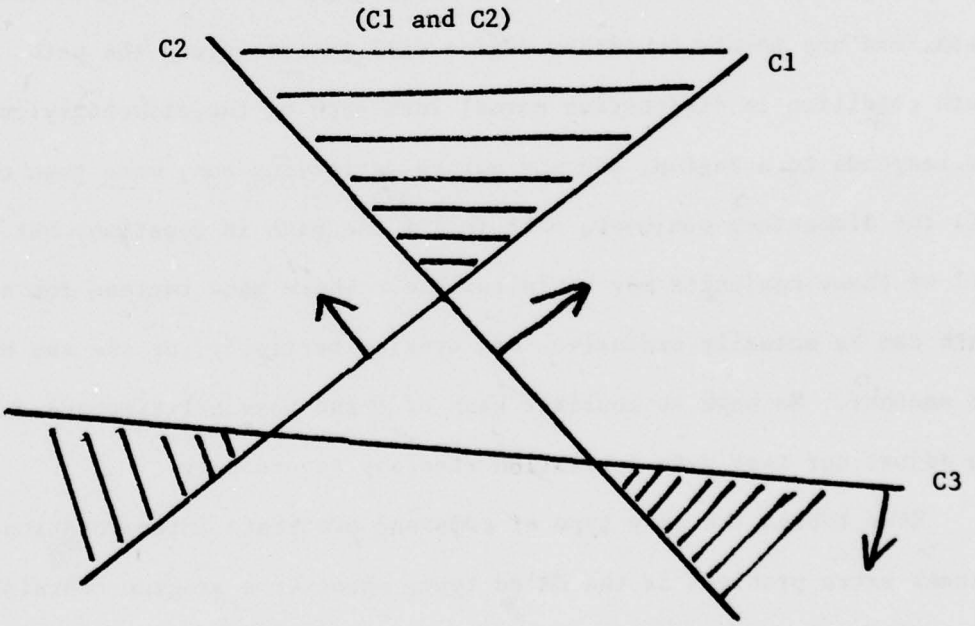


Fig. 5: Infeasible Path with Path Condition (C1 and C2 and C3)

One last problem which must be addressed is the case in which predicate interpretations contain array references with variable subscripts. In this case the constraint contains an Indeterminate Array Reference since the element of the array being constrained depends on the other input values. This is further complicated when these input variables are also constrained by the set of predicates.

This problem has to be attacked heuristically. Obviously the first goal has to be the assignment of specific values to those input variables which appear as subscripts in any of the constraints. The basic strategy in selecting values for these subscript variables is to produce a reduced system with a high probability of being solved. The values chosen obviously must meet the constraints on the subscript variables, and we also want to produce as simple a reduced system as possible. Of course, we must allow for an iterative solving capability in case the specific values chosen for the subscript variables produce an infeasible reduced system.

3.6 A Sample Program and Input Space Partitioning Diagram

The discussion in this section refers to the sample program in Figure 6, the list of paths and predicate interpretations in Figure 7, and the input space diagram in Figure 8.

In the example some explanation of the conventions and notation used is needed. In the sample program the code structure is a sequence of IF constructs. Since we have five IF constructs, each path is represented by a string of five letters, either T or E, where the first letter is associated with the first IF construct and so on. For example the path ETEET is a path flowing through the ELSE option of the first, third, and fourth IF constructs and through the THEN option of the second and fifth IF. Also the notation T---- represents the set of paths which execute the then option of the first IF construct; - stands for (T or E). In addition each path is assigned a number to relate it to the appropriate input space region in the diagram. Also to help interpret the diagrams each predicate has been assigned a boundary

form such as solid line, dashed line, etc. This clarifies the relation between each predicate and the corresponding border segments. The small arrows attached to the border segments indicate which region the border itself belongs to. These arrows will point to the region for which the border is a closed border. Of course, this example uses two input variables to allow a two-dimensional view of the partitioning of the input space. Even in this simple example we see that the partitioning of the input space is quite complicated. The concepts and phenomena displayed in this two-dimensional space will extend to the general case of higher dimensionality spaces.

This example program has five predicates, each a non-nested IF construct. Of course, the first predicate has only a single interpretation (see the list of interpretations in Figure 7), and this is reflected in the fact that this predicate leads to a single continuous border across the space. All feasible paths of the form T---- will correspond to regions above this border; all feasible paths of the form E---- will correspond to regions below the border. The second predicate has two interpretations $I \leq -2$ and $I \geq 2$. The former for paths of the form T----, and the latter for E---- paths. This can be seen as the two discontinuous vertical border segments for this predicate, both ending at the border for the first predicate. The fourth predicate, $G < -3$, should have two interpretations, but these two interpretations are degenerate. That is, by coincidence they reduce to the same interpretation, $J < -3$. This is reflected in the diagram as the single continuous border at $J = -3$.

The fifth predicate could lead to as many as 16 different interpretations, but in the input space only four border segments are actually produced. Because G = F -2*I + 2 reduces to G = J in statement (13), eight of the 16 possible interpretations disappear. The eight remaining interpretations are listed in Figure 7. We can see that four of these eight interpretations

do not appear in the input space since every path they affect is either infeasible or one on which the predicate is dominated.

Predicate three is probably the most interesting aspect of this program. It is the first example of an equality, and therefore not-equals, constraint that we have seen. Predicate three is represented in the diagram by the heavy solid line. We see that there are five feasible paths out of a total of 16 for which the equality is true. These are, top to bottom, regions 3, 11, 27, 19, and 17, and they each consist of only those points on the line, subject to the other constraints. Also, regions 7, 15, 31, 23, and 21 are split into two subregions by the line. These are the feasible paths for which the not-equals constraints is true and is not dominated.

We can see the phenomenon of predicate dominance in the fact that even though each path has to satisfy five constraints, many of the regions have less than five border segments. For example, region 14 has only two border segments since predicates 2, 3, and 5 are all dominated by C_1 AND C_4.

One last point is the apparent problem of infeasibility. With five non-nested IF constructs we can expect to have $2^5 = 32$ control paths. We can see that of these 32 control paths only 16 are feasible execution paths. Therefore a major problem will be that we might use a lot of our resources analyzing paths which in the end turn out to be infeasible, non-executable, and untestable. We need some way to recognize infeasible paths at an early stage of the testing process without doing a lot of processing. Even if we can't recognize all infeasibilities this early, any which we do find will save us a lot of wasted effort.

```
READ I,J;
C = I + 2*J - 1;

IF C > 6                                    P1 ————————————————————
    THEN D = C - I;
    ELSE D = C + I;
ENDIF;

IF D ≥ C + 2                                P2 ——————— — ———————
    THEN E = I;
    ELSE E = 3;
ENDIF;

F = 2*C - 3*J;

IF F = 0                                    P3 ▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬
    THEN G = F - 2*I + 2;
    ELSE G = J;
ENDIF;

IF G < -3                                   P4 ——————— — — ———————
    THEN G = 0;
    ELSE;
ENDIF;

IF 2*D + 3*E ≥ F - G + J                    P5 — — — — — — — — — —
    THEN H = I;
    ELSE H = J;
ENDIF;

WRITE H;
```

Fig. 6: Sample Program with 5 Predicates

#	PATH	P1	P2	P3	P4	P5	DOM. PREDS.
1	TTTTT	$I+2J > 7$	$I \leq -2$	$2I+J = 2$	$J < -3$	$I+2J \geq 0$	infeas.
2	TTTTE	"	"	"	"	$I+2J < 0$	infeas.
3	TTTET	"	"	"	$J \geq -3$	$I+3J \geq 0$	1,4,5
4	TTTEE	"	"	"	"	$I+3J < 0$	infeas.
5	TTETT	"	"	$2I+J \neq 2$	$J < -3$	$I+2J \geq 0$	infeas.
6	TTETE	"	"	"	"	$I+2J < 0$	infeas.
7	TTEET	"	"	"	$J \geq -3$	$I+3J \geq 0$	4,5
8	TTEEE	"	"	"	"	$I+3J < 0$	infeas.
9	TETTT	"	$I > -2$	$2I+J = 2$	$J < -3$	$-2I+2J \geq -9$	infeas.
10	TETTE	"	"	"	"	$-2I+2J < -9$	infeas.
11	TETET	"	"	"	$J \geq -3$	$-2I+3J \geq -9$	4,5
12	TETEE	"	"	"	"	$-2I+3J < -9$	infeas.
13	TEETT	"	"	$2I+J \neq 2$	$J < -3$	$-2I+2J \geq -9$	infeas.
14	TEETE	"	"	"	"	$-2I+2J < -9$	2,3,5
15	TEEET	"	"	"	$J \geq -3$	$-2I+3J \geq -9$	4
16	TEEEE	"	"	"	"	$-2I+3J < -9$	2,3
17	ETTTT	$I+2J \leq 7$	$I \geq 2$	$2I+J = 2$	$J < -3$	$5I+2J \geq 0$	1,2,5
18	ETTTE	"	"	"	"	$5I+2J < 0$	infeas.
19	ETTET	"	"	"	$J \geq -3$	$5I+3J \geq 0$	1,5
20	ETTEE	"	"	"	"	$5I+3J < 0$	infeas.
21	ETETT	"	"	$2I+J \neq 2$	$J < -3$	$5I+2J \geq 0$	
22	ETETE	"	"	"	"	$5I+2J < 0$	1,3,4
23	ETEET	"	"	"	$J \geq -3$	$5I+3J \geq 0$	5
24	ETEEE	"	"	"	"	$5I+3J < 0$	infeas.
25	EETTT	"	$I < 2$	$2I+J = 2$	$J < -3$	$2I+2J \geq -9$	infeas.
26	EETTE	"	"	"	"	$2I+2J < -9$	infeas.
27	EETET	"	"	"	$J \geq -3$	$2I+3J \geq -9$	4,5
28	EETEE	"	"	"	"	$2I+3J < -9$	infeas.
29	EEETT	"	"	$2I+J \neq 2$	$J < -3$	$2I+2J \geq -9$	1,3
30	EEETE	"	"	"	"	$2I+2J < -9$	1,3
31	EEEET	"	"	"	$J \geq -3$	$2I+3J \geq -9$	
32	EEEEE	"	"	"	"	$2I+3J < -9$	2,3

Fig. 7: Paths and Predicate Interpretations

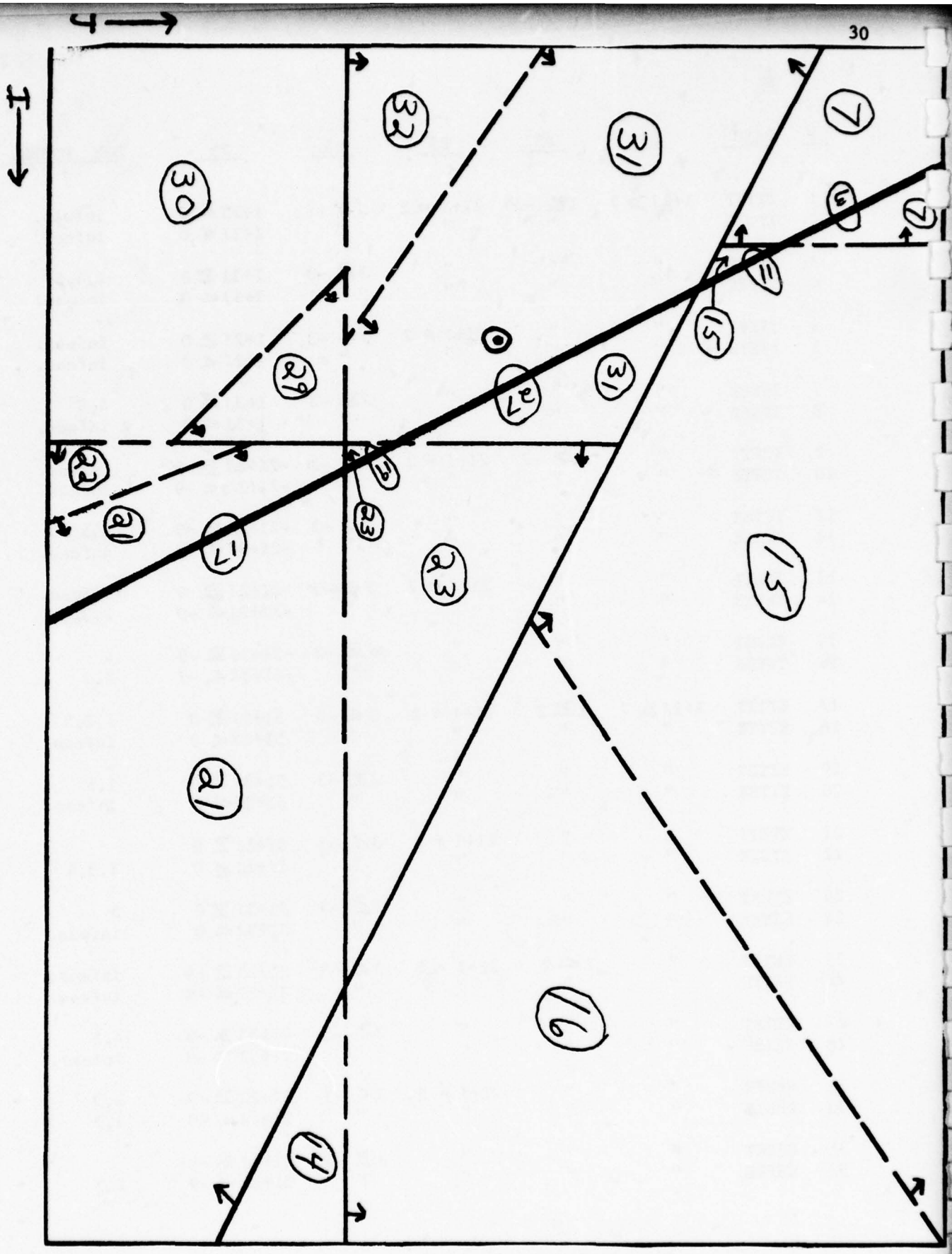


Fig. 8: Sample Input Space Partitioning Diagram

CHAPTER IV PRELIMINARY RESULTS

4.1 Error Classification

In program testing we define an error as any discrepancy between what the program computes for some set of input values and what we expect it to compute. Therefore, a natural error classification scheme is one based on how the error affects the input-output relationship. Almost all error classification schemes appearing in the literature are based on the programming mistake which caused the error, but in this work, the observable effect of the error is more important.

In program testing the natural unit of testing is a program path or equivalently the region of the input space which executes a specific path. The computation performed along any path consists of two parts, a set of transformation formulae and a path condition compound predicate. The transformation formulae define the computation on the path which produces the output variables' values, and the conjuncted path condition provides the definition of the subset of the input space points which will follow the particular path. Therefore, our error classification scheme will be based on which of these components is in error and how that error's effects can be found. In order to be able to talk about errors and their effects we have to assume that there exists a program which is correct for performing the same task that the program which we are testing is supposed to perform. This "Correct Program" might or might not be the program we are testing. We need this concept of a correct program to enable us to discuss how various classes of errors which might exist in the program being tested will cause the computations performed by the tested program to be different from that which should be performed or would be performed by the correct program.

One problem which is inherent to the process of program testing is that of Coincidental Correctness. The basis of the program testing

process is that from the observed correctness of the program over a small set of points (test points) we infer that the program is correct over a much larger set of points (an input space region). If the observed test points produced the correct output values using incorrect computations the basis of our inference is destroyed. In other words the program produced the correct output at the test points because the function it computed just happens to coincide with what the correct program would compute at those points. This phenomenon is a basic obstacle to achieving completely reliable program testing. However, in a practical sense, our method of choosing many widely separated test points for a particular path greatly reduces the already low probability of producing several coincidentally correct output values for a path.

Any error can change the transformation a path performs, change the borders of the input space area over which the path is active, or cause both the transformation and the borders to be in error. So the first class of errors to discuss is that class in which the borders of the input space region are correct, or in other words the path's domain is the same as the corresponding domain in the correct program's input space partitioning. The path operates on the correct subset of the input space, but it computes the wrong function over the area. This type of error is called a Transformation Error. In this case, testing would seem to be easy since we can find this error by testing any single point in the path's domain. However, the possibility of coincidental correctness affects this result. Theoretically, we would have to test every point in a region to ensure the complete correctness of the transformation. A transformation error exists when the value of a variable is computed incorrectly in an assignment statement. Further, this variable must be one which affects the calculation of

an output variable but which is not used, either directly or indirectly, in any predicate along the path.

A second class of errors are those in which the transformation is correct, but an error in a predicate interpretation causes a border to be shifted from its position in the correct input space partitioning structure. Essentially the points in the area between the correct position of the border and the shifted position of the border will follow the wrong path. Three cases of a shifted border must be considered. In the first case the border has shifted from the correct border to enlarge the region, and some points execute on this path which should follow another path. The second case is the opposite where the region has been reduced, and some points in another region should follow this path. The final case is a combination of the first two where some points are missing from the region and other points in the region should be in other regions. This is the case where the incorrect shifted border intersects what should be the correct border. This class of error can be caused by an incorrect predicate or by an assignment statement error which affects a predicate but is not used in the computation of any output value. Of course, a single assignment statement error can produce multiple border shifts.

A third error class is when both the border(s) shift and the transformation is incorrect. This occurs when an assignment statement error affects a program variable which is used both in a predicate and in computing the output values. This class of errors can be found similarly to the second class described above since it can be viewed as the simultaneous occurrence of a class 1 and a class 2 error. In general, multiple errors on the same path cause no problems except in the rare case where one error exactly cancels the other.

There is one final class of errors which we must consider called

Missing Path Errors. In the first three classes of errors which we defined the paths involved in the error exist in both the correct program and in the one we are testing. In this final class of errors there is a path in the correct program which does not even exist in the program being tested. A subregion of the input space region we are testing should be a separate region by itself, associated with a completely new path. So, this type of error is similar to classes two and three in that there are points in the region which belong in another region. The difference lies in the fact that this other region does not exist in the current incorrect input space structure. For a missing path error, the subregion which should be executed by the missing path can vary in size from a large fraction of the region down to a single point. Also there is no indication of where in the region this subregion might be.

So in conclusion we have an error classification scheme based on the error's observable effect on the input space structure and the computational transformations.

CLASS	DOMAIN	TRANSFORMATION
1	correct	incorrect
2	incorrect (border(s) shifted)	correct
3	incorrect (border(s) shifted)	incorrect
4	incorrect (should be 2 regions)	correct

4.2. Domain Testing for Linear Borders in Two Dimensions

In this section we introduce a testing strategy which will guarantee that any error of classes two and three will be found by one or more of the test points. The present discussion assumes linear borders

in a two dimensional space. Later sections will extend this analysis to higher dimensional spaces and to non-linear borders.

In general a domain is a convex polytope bordered by segments of hyperplanes (lines in the two dimensional case). These borders can be open or closed, but we can replace an open border represented by

$$A_1X_1 + \ldots + A_kX_k < B$$

by an equivalent closed border represented by

$$A_1X_1 + \ldots + A_kX_k \leq B - \varepsilon$$

where ε *is the smallest positive number in the data representation.* So the problem is to select a small set of points which will allow us to recognize any border which has been shifted from its correct position.

We first analyze an idealized situation in which the input space is continuous, and therefore we can make ε as near to zero as we want. We select two points on each of the border line segments and a third point for each border located between the first two but ε outside the region. Figure 9 demonstrates this set of points for a domain, and Figure 10 shows a single border.

If the program produces correct results at the three selected test points then we know that the border is correct. So we can find all border shift errors by testing at most 3*K points where K is the number of border segments defining the domain. We know that the correct border must pass through or above the two ON test points, X_1 and X_2, and under the OFF test point, X_3. Because of the alternating sequence that means that if the given border is incorrect, the correct border must intersect the given border twice, in each ON-OFF and OFF-ON interval. Since no line exists which can intersect a given line segment in two points the given one must be correct. Of course, the correct border might pass between the given border and the OFF test point without

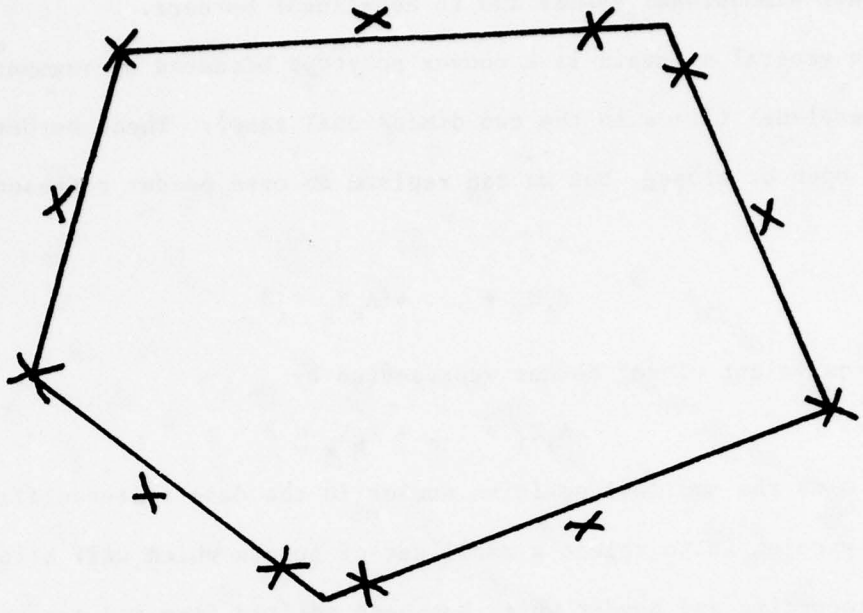


Fig. 9: Domain Testing for A Single Domain

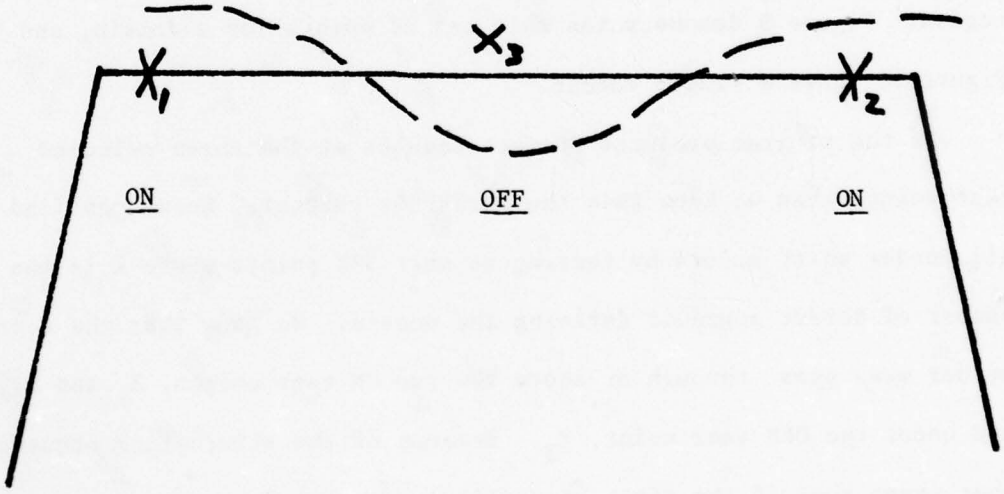


Fig. 10: Domain Testing for A Single Border

of this happening also approaches zero. This is true for each border of the region.

In the last section, three types of border shifts were identified, and we can see how our set of three test points can catch each of these types of shifts. In Figures 11-14, the solid lines are the given borders, the dashed lines are what the correct borders would be, and the X's are the test points.

Figure 11 shows the case in which the border has shifted to shrink the region. When we test X_3 the transformation for the region R' will be applied while the transformation for R should be used. This difference will signal the error. In Figure 12, the border has shifted to enlarge the region R. I this case both X_1 and X_2 will be computed using the transformation for region R while they should be part of R'. In Figure 13, we can see that point X_1 should be computed as a point in R' but it will be computed with the R region transformation.

The preceding discussion assumed a continuous input space and an ϵ approaching zero. In reality the input space is discrete, and there is a definite limit to how small we can make ϵ. The question is how does this affect our ability to find border shifts. We will see that a finite ϵ means that we won't be able to catch certain border shifts of magnitude less than ϵ.

The three test points for a particular border tell us only that the correct border lies on or above the two ON points and below the OFF point. When the OFF point is a finite distance ϵ from the border there is a set of possible correct borders of which the given one is a member. This ϵ-set of borders can be defined precisely as follows. Construct a line segment between the OFF test point (open end) and each ON test point (closed end). The correct border must intersect each of these line segments, and therefore this is the ϵ-set of possible

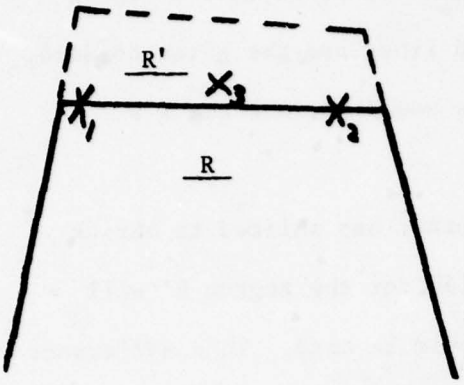


Fig. 11: Border Shift Reducing Domain

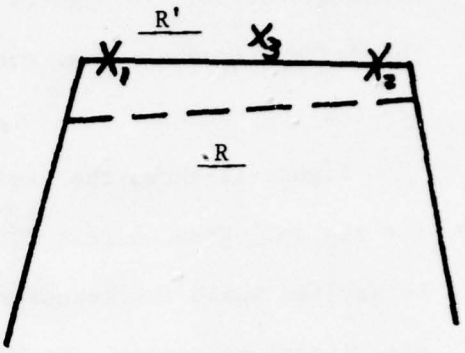


Fig. 12: Border Shift Enlarging Domain

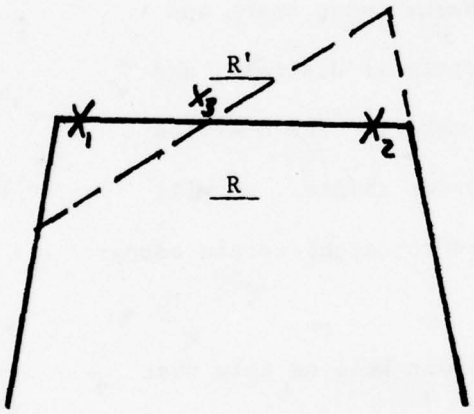


Fig. 13: Intersecting Border Shift

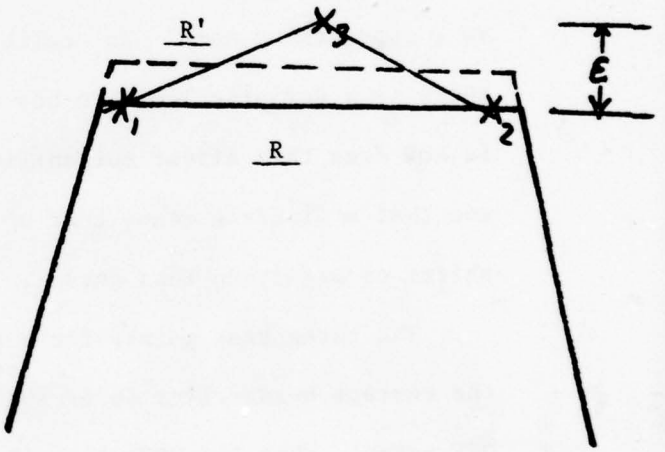


Fig. 14: Border Shift Less Than ε

correct borders. So in a real discrete space our testing strategy tells us that we can detect any shift greater than ϵ. Since ϵ is the smallest positive number in the data representation being used, the probability of a shift this small is miniscule and the effect is insignificant. This case is diagrammed in Figure 14.

4.3 Missing Path Errors

A missing path error is the case where the region under test should be two regions, one for the path under test and one for a path which does not even exist in the program being tested. This differs from previously discussed domain errors in that the subregion in error can be located anywhere within the region and can be as small as a single point.

The missing path occurs because there should be another predicate on the path under test thus forming two paths. Typically, this is the case when a special condition has been omitted. We can classify missing path errors according to the type of predicate which has been omitted. We will be able to show that the domain testing strategy developed in the last section will catch missing path errors except one class which we call Missing Path Errors of Reduced Dimensionality.

Assuming that the missing predicate is simple we can form three classes: the inequality predicates ($<$, $\leq$, $>$, $\geq$), the equality predicate ($=$), and the not-equals predicate ($\neq$). Any of these prediates defines a hyperplane cutting the region under test. For the inequality predicates the subregion for the missing path will be that section of the region lying on one side or the other of the hyperplane. For the not-equals predicate the subregion will be the entire region minus the hyperplane. In either case one or more of our domain testing points will fall in the subregion, and we will discover the missing path error. However, when the missing special condition is an equality, the subregion where

the error can be discovered is just a hyperplane cutting through the region. Since we have no indication of where this hyperplane should be and the hyperplane itself forms a subregion of zero-measure in comparison to the entire region there is no effective method of finding it. In Figures 15-17, R' specifies the subregion for the missing path and (X) specifies a domain test point which will catch the error.

So in summary, missing path errors of reduced dimensionality caused by an omitted predicate are impossible to discover by testing. This poses a basic limit for any program testing technique. However, many types of missing path errors will be found without the need for extra test points.

4.4. Domain Testing of Linear Borders in Higher Dimensions

In this section we generalize the domain testing result for any N dimensional space. We will see that the result generalizes completely, and the number of test points needed to implement the Domain Testing Strategy is bounded by $K*(N+1)$ where N is the dimensionality of the space and K is the number of hyperplane segments making up the border of the region. This is only an upper bound since an edge or extreme point can be used to test more than one border.

In an N dimensional space each border is a hyperplane which can be defined by N linearly independent points. Also, two hyperplanes of an N-dimensional space intersect in a single hyperplane of an N-1 dimensional space, which can contain at most N-1 linearly independent points. To test a border we select N linearly independent points on the border. We also select a single point off the border whose projection on the border is a point which is a convex combination of the set of ON points. So for each border we have N linearly independent ON points and a single OFF point. This set of test points is as effective as the three points in the two dimensional case.

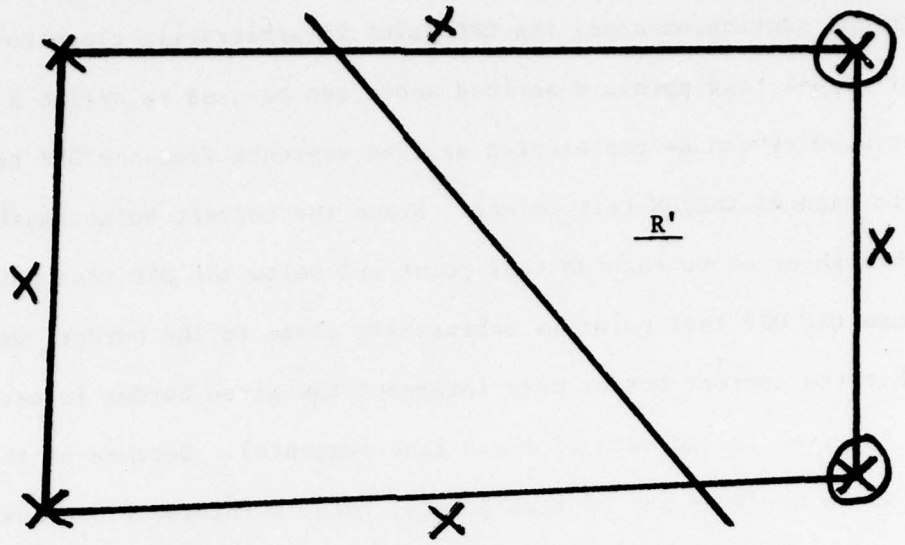


Fig. 15: Missing Path Error for An Inequality Predicate

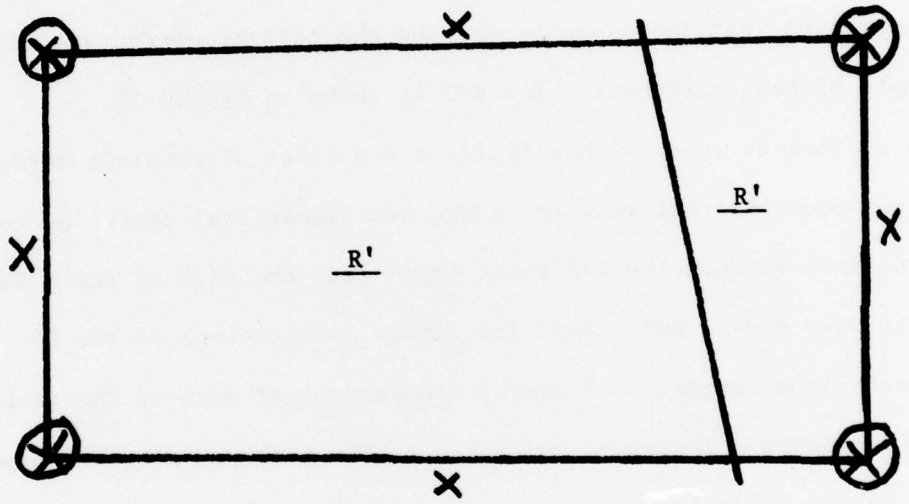


Fig. 16: Missing Path Error for A Not-equals Predicate

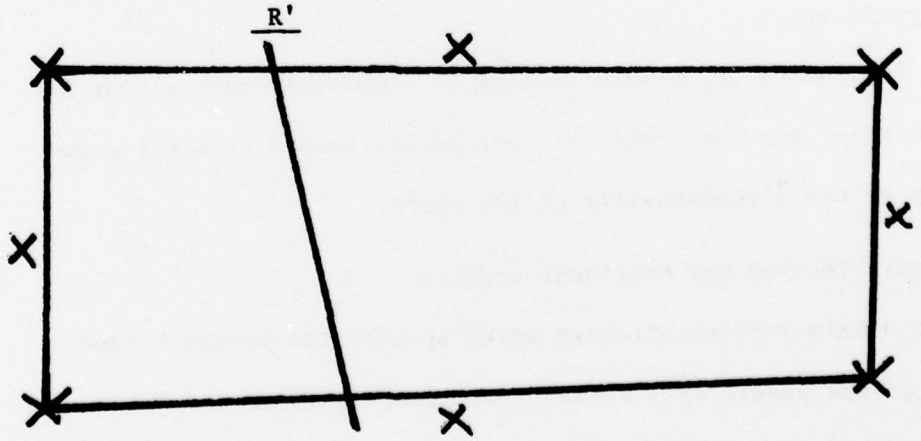


Fig. 17: Missing Path Error of Reduced Dimensionality for An Equality Predicate

In the continuous space the OFF point is arbitrarily close to the border. The set of N+1 test points described above can be used to define N OFF-ON intervals which can be represented as line segments from the OFF test point to each of the ON test points. Since the correct border must pass through or above each ON test point and below the OFF test point and since the OFF test point is arbitrarily close to the border, we know that the correct border must intersect the given border in each OFF-ON interval (along each of the N line segments). Because of the way in which we chose the ON test points, these N intersections must be linearly independent. From this and the fact that the intersection of two hyperplanes can contain at most N-1 linearly independent points, we can conclude that the given border and the correct border are identical. An example of the construction for N=3 is shown in Figure 18.

In a discrete space with a finite ε the ε-set of possible borders can be defined constructively as in the two dimensional case. We construct a line segment between the OFF point (open end) and each of the N ON points (closed end). Because of the linear independence of the ON points each line segment is linearly independent of each of the other N-1 line segments. Therefore, a member of the ε-set of possible hyperplane borders can be uniquely defined as a hyperplane segment which intersects each of the N line segments. Again, since ε is so small this ε-error is insignificant.

So in conclusion, domain testing is completely general for any linear border, and the number of test points needed at worst grows linearly as the dimensionality of the space.

4.5. Domain Testing for Nonlinear Borders

The domain testing strategy works so well for linear borders because of the predictable behavior of linear borders and the way they can intersect. The general class of nonlinear forms does not exhibit a similar predictability, and therefore domain testing

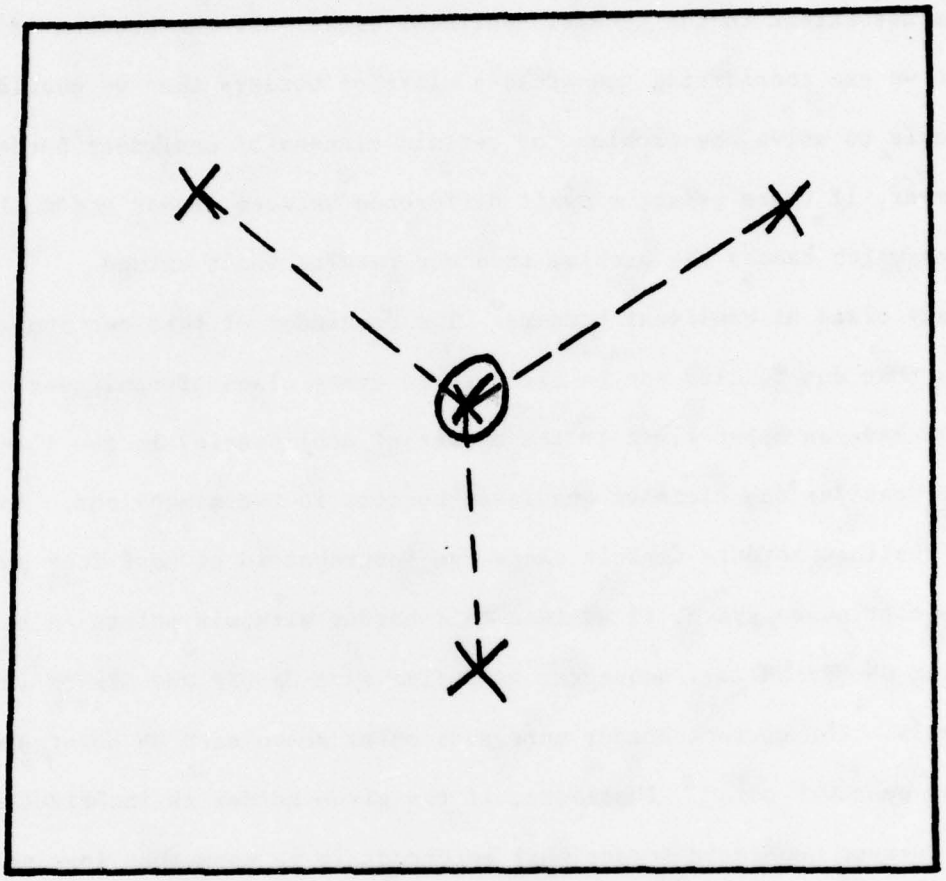


Fig. 18: Domain Testing for A Border in A Three Dimensional Space

will not be completely effective for the complete class of nonlinear borders. This is because no matter how many OFF-ON and ON-OFF intervals we define along the nonlinear border there exists another border with a higher degree of nonlinearity which can intersect the given border an arbitrarily high number of times (at least once in each interval).

A basic question which must now be addressed is why this result does not extend to the general nonlinear class. If the problem is that we are considering too broad a class of borders then we should be able to solve the problem for certain classes of nonlinear borders. However, if there exists a basic difference between linear and nonlinear forms which causes the problem then our results won't extend to any class of nonlinear borders. The remainder of this section will show that our results can be extended to every class of nonlinear borders if we have an upper limit to the degree of nonlinearity in the class.

Consider the class of quadratic borders in two dimensions. Any two distinct borders of this class can intersect in at most four points. In a continuous space, if we test this border with six points in alternating ON-OFF-ON .... sequence, we define five ON-OFF and OFF-ON intervals. The correct border must pass on or above each ON point and below each OFF point. Therefore, if the given border is incorrect, the correct quadratic border must intersect it in more than four points. So we can conclude that the given border is correct (see Figure 19). For discrete spaces the ϵ-error analysis is analogous to the linear case.

For any nonlinear class we determine the maximum number of intersections which two distinct members of the class can form; call it K. We then select K+2 test points in alternating ON-OFF-ON .... sequence along the border. These K+2 points define K+1 intervals, each of which

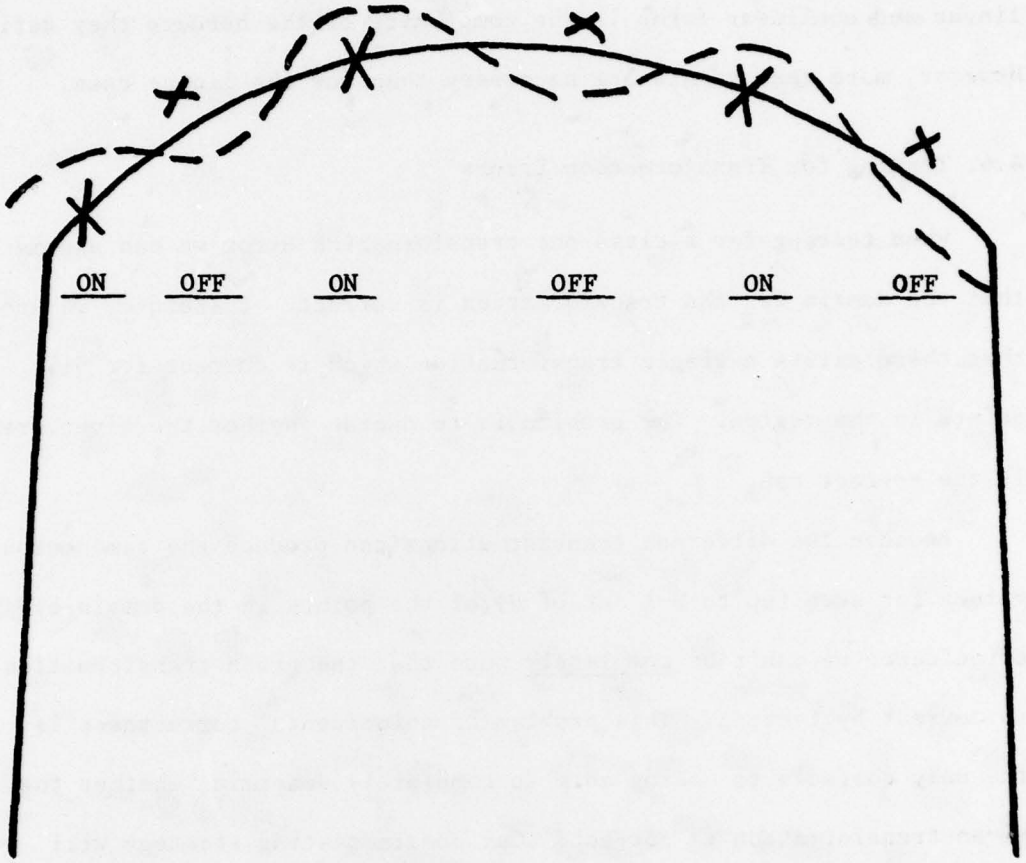


Fig. 19: Domain Testing for A Nonlinear Border

must contain an intersection between the given border and the correct border. The impossibility of more than K intersection points allows us to conclude that the given border is correct.

So in conclusion, we have shown that domain testing is effective for nonlinear borders under the reasonable assumption of a maximum degree of nonlinearity. Therefore, the only important difference between linear and nonlinear forms is the complexity of the borders they define. However, more test points are necessary than for the linear case.

4.6. Testing for Transformation Errors

When testing for a class one transformation error we can assume that the domain for the transformation is correct. Therefore, we know that there exists a single transformation which is correct for all points in the region. The problem is to decide whether the given transformation is the correct one.

Because two different transformations can produce the same output values for some (up to N-1 out of N) of the points in the domain by coincidence we can't be completely sure that the given transformation is correct by testing. This problem of coincidental correctness is the only obstacle to being able to completely determine whether the given transformation is correct. Our domain testing strategy will test N*K points in the given domain where N is the dimensionality of the space and K is the number of border segments. If the output values computed for these points are correct then either the transformation is correct or it is coincident with the correct transformation at each of the N*K widely separated points. Because of the low probability of this happening we choose not to test any additional points in the region beyond these N*K boundary points. Coincidental correctness is an inherent risk associated with testing using finite samples.

CHAPTER V Future Work

In the current phase of this research the last major task is to develop an algorithm to generate real test data meeting the criteria described by the domain testing strategy. In developing this algorithm we are assuming that the set of predicate interpretations for the program control path being tested has already been generated during the symbolic execution of the path. The final output of the algorithm will be a complete set of domain test values for a feasible path or an indication that the specified program control path is infeasible and cannot be tested. The algorithm is broken down into four distinct phases.

Phase 1 is essentially a preprocessing phase in which the predicates are separated into three classes (equalities, inequalities, and not-equals). Also a pair of min-max constraints are added for each input variable representing the minimum and maximum values for a variable of that particular data type. In the second phase all equality predicates are processed. Each linearly independent equality predicate constrains the value of a single input variable and can be used to eliminate that variable from the entire set of predicates. Therefore, during the second phase of the algorithm the dimensionality of the problem can be reduced considerably. The third phase of the algorithm uses linear programming techniques to generate a set of test values for the outer border of the domain as defined by the domain testing strategy. In the fourth and final phase of the algorithm, test values are selected to test the interior borders which are generated by the not-equals predicates. An important problem is how best to apply the standard linear programming techniques in phases 3 and 4.

Another area which will be investigated is that of an "Optimal Domain Testing Strategy". Since the domain testing strategy proceeds a single path at a time, each border in the input space structure will be tested many times. This results in the generation of more test points than are really necessary. An optimal domain testing strategy would define the minimum set of points needed to test the borders of the input space and would provide a strategy to generate only those points.

The problem of coincidental correctness has been identified as an inherent theoretical limitation to any testing procedure. However, many practical aspects of coincidental correctness need to be addressed. Once we gain insights into how coincidental correctness can occur for a specific path, we can determine the path characteristics which correlate with a high likelihood of coincidental correctness. We also need to investigate practical techniques which can be used to increase our ability to recognize instances of coincidental correctness.

The analysis of the domain testing strategy for nonlinear borders as described in this report considers only nonlinear borders in two dimensions. This result must be extended to nonlinear borders in a general N-dimensional input space. In particular, we need to determine how quickly the set of required test points grows as we test nonlinear borders in higher and higher dimensional spaces.

The complications caused by the reality of a discrete input space were briefly discussed, but all the ramifications of the discreteness problem need to be analyzed. In these studies the input space will be modeled as a discrete lattice of representable points. In particular, we need to know how a discrete input space will affect the actual implementation of the domain testing strategy. Also, various pathological cases in which a domain or border contains only a small number of representable

points will be analyzed. In addition, the computational complexity of completely solving these discretization problems will be investigated.

The domain testing strategy offers an alternative to exhaustive testing in which a major reduction in the number of test values needed is achieved with a minimal loss of testing effectiveness. However, the required set of test points for a nontrivial program may still be too large for practical testing situations. Ultimately, we would like to develop a flexible strategy which will maximize testing effectiveness over a wide range of cost levels. To this end, we plan to investigate extensions to the domain testing strategy which further reduce the number of required test points. A "Partial Domain Testing Strategy" is one in which only a subset of the points required by the domain testing strategy is generated for each path. The problem here is to determine the best ordering of the test points for a domain, based on their testing effectivenss.

The domain testing strategy assumes that each path will be tested. However, the number of paths in a program can be too large for practical purposes. A "Path Selection Strategy" is needed which will select program paths for testing, based on some measure of each path's testing value in relation to the entire program. Since a programming error in any statement will, in general, affect many paths in the program, we expect that an appropriate path selection strategy can be used to generate a highly effective set of test data at a reasonable cost.

In future phases of this research we foresee the need of a measure of testing effectiveness. Program testing cannot be completely effective, so we must be able to provide some estimate of the effectiveness of the particular set of test data selected. This measure could be based upon how well the selected set of test paths represents the complete set of program paths and on how completely we have tested each path.

At present we are assuming that a user who knows exactly what the program should compute for any specific test case will decide whether each test case is correct or not. One possible avenue for future research would be to automate this process by using some form of specification written by the user to determine the correctness of each particular test case. An effective procedure for determining the correctness of the test cases would be extremely important in reducing the total cost of program testing.

The long term goals of research in the area of software quality assurance is a set of techniques which will help us to produce more reliable software. The study of one of these techniques does not preclude the use of any other technique. In fact we expect these techniques to be complementary rather than competitive. The long term goal of our research is the integration of the program testing methodology with the more theoretical approach of program verification. It is hoped that each technique will overcome some of the obstacles encountered in the use of the other technique. This type of hybrid approach should be more successful than either technique implemented separately.

LIST OF REFERENCES

BOEHB73 Boehm, B. W., "Software and Its Impact: A Quantitative Assessment", Datamation, Vol. 19, No. 5, May 1973, 48-59.

BOYER75 Boyer, R. W., Elspas, B., and Levitt, K. N., "SELECT -- A Formal System for Testing and Debugging Programs by Symbolic Execution", PROCEEDINGS - 1975 International Conference on Reliable Software, 234-245.

CLARL75 Clarke, L. A., "A System to Generate Test Data and Symbolically Execute Programs", IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, September 1976, 215-222.

ELSPB72 Elspas, B., Levitt, K. N., Waldinger, R. J., and Waksman, A., "An Assessment of Techniques for Proving Program Correctness", ACM Computing Surveys, Vol. 4, No. 2, June 1972, 97-147.

GABOH76 Gabow, H. N., Maheshwari, S. N., and Osterweil, L. J., "On Two Problems in the Generation of Program Test Paths", IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, September 1976, 227-231.

GOODJ75 Goodenough, J. B. and Gerhart, S. L., "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1972, 156-173.

HETZW73 Hetzel, W. C., ed., Program Test Methods, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1973.

HOWDW75 Howden, W. E., "Methodology for the Generation of Program Test Data", IEEE Transactions on Computers, Vol. C-24, No. 5, May 1975, 554-560.

HOWDW76 Howden, W. E., "Reliability of the Path Analysis Testing Strategy", IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, September 1976, 208-215.

HOWDW77 Howden, W. E., "Symbolic Testing and the Dissect Symbolic Evaluation Systems", IEEE Transactions on Software Engineering, Vol. SE-3, No. 4, July 1977, 266-278.

HUANJ75 Huang, J. C., "An Approach to Program Testing", ACM Computing Surveys, Vol. 7, No. 3, September 1975, 113-128.

IEEE75 IEEE, PROCEEDINGS - 1975 International Conference on Reliable Software, held 21-23 April 1975 in Los Angeles, California.

KINGJ75 King, J. C., "A New Approach to Program Tesitng", PROCEEDINGS - 1975 International Conference on Reliable Software, 228-233.

KINGJ76 King, J. C., "Symbolic Execution and Program Testing", Communications of the ACM, Vol. 19, No. 7, July 1976, 385-394.

MILLE75 Miller, E. F., "Methodology for Comprehensive Software Testing", General Research Corporation Report No. RADC-TR-75-161, also NTIS No. AD/A013-111, June 1975.

MILLE74 Miller, E. F. and Paige, M. R., "Automatic Generation of Software Testcases", Eurocomp Conference Proceedings, 1974, 1-12.

RAMAC76 Ramamoorthy, C. V., Ho, S. F., and Chen, W. T., "On the Automated Generation of Program Test Data", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976, 293-300.

TANEA76 Tanenbaum, A. S., "In Defense of Program Testing or Correctness Proofs Considered Harmful", ACM SIGPLAN Notices, Vol. 11, No. 5, May 1976, 64-68.

YEHR76 Yeh, R. T., ed., "Special Issue: Reliable Software I: Software Validation", ACM Computing Surveys, Vol. 8, No. 3, September 1976.

YEHR76A Yeh, R. T., ed., "Special Issue: Reliable Software II: Fault-Tolerant Software", ACM Computing Surveys, Vol. 8, No. 4, December 1976.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 78-1014 ✓ 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* A FINITE DOMAIN-TESTING STRATEGY ✓ FOR COMPUTER PROGRAM TESTING	5. TYPE OF REPORT & PERIOD COVERED Interim ✓
	6. PERFORMING ORG. REPORT NUMBER (OSU-CISRC-TR-77-13) ✓
7. AUTHOR(s) Edward I. Cohen and Lee J. White	8. CONTRACT OR GRANT NUMBER(s) AFOSR -77-3416 ✓
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer & Information Science Research Center ✓ The Ohio State University Columbus, Ohio 43210	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F , 2304/A2 .
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM Bolling AFB, Washington, DC 20332	12. REPORT DATE August 1977 ✓
	13. NUMBER OF PAGES 55
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)* UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

Approved for public release; distribution unlimited.

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Program testing continues to be a practical approach to software validation, however the strategies currently being used lack a solid analytical foundation. The goal of this research is to analyze the program testing process, develop a strategy to maximize its effectiveness, and identify its limitations.

A program can be viewed as a complex mapping from an N-dimensional space of input variables to an M-dimensional space of output variables.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

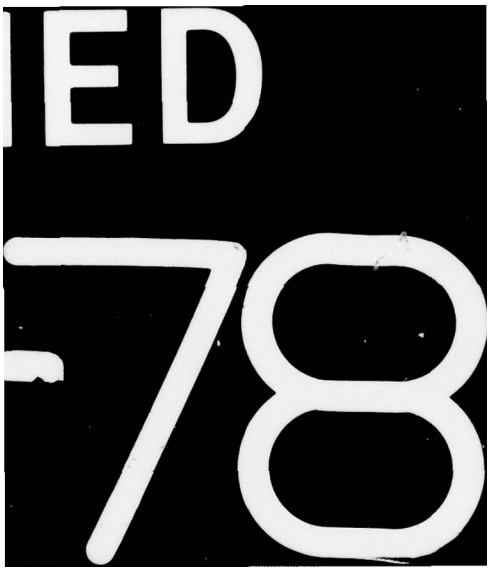
UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

20.

In the testing process the correctness of the program over a domain of this input space is inferred from its observed correctness on a small set of "well-chosen" test values for that domain. The Domain Testing Strategy is used to determine the necessary set of test values and is shown to be successful for all types of errors except a small subclass called "Missing Path Errors of Reduced Dimensionality". The domain testing strategy is developed for both continuous and discrete input spaces and for both linear and nonlinear predicates.

The only completely effective testing strategy is an exhaustive test which is totally impractical. The domain testing strategy offers a major reduction in the high cost of computer program testing with a minimal loss of testing effectiveness.


ED
78